# Blockin: Multi-Chain Sign-In Standard with Micro-Authorizations

MATT DAVISON, Virginia Tech, United States
KEN KING, Virginia Tech, United States
TREVOR MILLER, Virginia Tech, United States

The tech industry is currently making the transition from Web 2.0 to Web 3.0, and with this transition, authentication and authorization have been reimagined. Users can now sign in to websites with their unique public/private key pair rather than generating a username and password for every site. However, many useful features, like role-based access control, dynamic resource owner privileges, and expiration tokens, currently don't have efficient Web 3.0 solutions. Our solution aims to provide a flexible foundation for resource providers to implement the aforementioned features on any blockchain through a two-step process. The first step, authorization, creates an on-chain asset which is to be presented as an access token when interacting with a resource. The second step, authentication, verifies ownership of an asset through querying the blockchain and cryptographic digital signatures. Our solution also aims to be a multi-chain standard, whereas current Web 3.0 sign-in standards are limited to a single blockchain.

Additional Key Words and Phrases: blockchain, security, authorization, smart contracts, access control, algorand, sign-in, authentication, nft

**ACM Reference Format:**
Matt Davison, Ken King, and Trevor Miller. 2022. Blockin: Multi-Chain Sign-In Standard with Micro-Authorizations. 1, 1 (November 2022), 10 pages. https://doi.org/XXXXXXX.XXXXXXX

## 1 INTRODUCTION

The backbone of the Internet is based on micro-authorizations, login details, and giving users certain permissions for websites and servers. Web 2.0 provided us with many authorization technologies, such as OAuth 2.0, access levels, JSON Web Tokens, and many more ways for parties to give permissions to their users. We are entering the period of Web 3.0 (the term for the new era of computing using blockchain technology and cryptocurrencies), and one of the new paradigms of Web 3.0 is users now connecting and logging in to all websites directly with their single, unique public/private key pair rather than generating a username/password combination for every site they visit. Because of this, many of the features of Web 2.0 authentication technologies previously mentioned are not directly translatable to Web 3.0 in a user-friendly or efficient manner. We are particularly interested in the following features that haven't translated well for this paper: role-based access control, subscriptions/expiring authorizations, and dynamic privileges for the resource owner. These are currently not natively possible in Web 3.0 without centralized storage defining roles and permissions of users because the process for authenticating public/private key pairs is the exact same for any pair (e.g. there is natively no "admin" keys or anything similar). We also make the observation that there is no universal, multi-chain Web 3.0 sign-in standard yet. There have been attempts at sign-in standards for specific blockchains, such as Sign in with Ethereum, but there isn't a standard that natively supports all blockchains.

Authors' addresses: Matt Davison, Virginia Tech, Blacksburg, United States, mattd7@vt.edu; Ken King, Virginia Tech, Blacksburg, United States, kking935@vt.edu; Trevor Miller, Virginia Tech, Blacksburg, United States, trevormil2001@vt.edu.

2022. XXXX-XXXX/2022/11-ART $15.00
https://doi.org/XXXXXXX.XXXXXXX

## 2 RELATED WORKS

The best solution we have seen for role-based access tokens is the use of non-fungible tokens (NFTs) where one grants access or proves ownership only if a user owns a certain NFT, as proven by the public blockchain. Some example implementations and use cases include patents, website certificates, smart grid authorization for IoT, and digital certifications [[Bamakan et al. 2022], [Kamboj et al. 2021], [Al-Bassam 2017], [Zhong et al. 2021]]. However, this solution historically doesn't perform well with regard to cost, speed, and scalability. Costs of deploying a smart contract on Ethereum currently average over $1000, and the time for transactions to be processed can range from a couple of minutes to over an hour. For everyday use cases such as logging in to a site or requesting data from a server, this is just not good enough. As for expirations and dynamic privileges for the resource owners, there have been implementations that have attempted to program this logic into smart contracts themselves, but these also run into the same scalability, cost, and speed bottlenecks as described above.

Another interesting solution to this problem was outlined in the OAuth 2.0 using Blockchain Tokens paper. This paper proposed a combination of Web 2.0 and Web 3.0 authentication technologies in order to provide role-based access and customizable permissions. First, the resource server would create a non-fungible token (NFT) that outlined the access permissions. When a user first requests access to that resource, they would transfer the NFT to their Ethereum wallet, and whenever the user requests access to that resource in the future, they can just verify on the public blockchain that the user owns the NFT issued by them and grant them access [[Fotiou et al. 2020]].

Our solution does not aim to replace the public/private key standard but rather build on top of it to expand its functionality for additional use cases. Our goal is to combine the previously mentioned solutions, other Web 3.0 technologies (such as smart contracts and digital assets), and the existing public/private key standard to provide a multi-chain, general interface for the functionality needed for role-based access, expiring tokens, and dynamic resource privileges. In particular, we hope to improve upon the speed, scalability, and cost of the existing solutions while also introducing new features such as freezing, expirations, and clawbacks of tokens. Our interface also provides the ability to program logic into the token's smart contract which can open up any possibility that can be programmed into a Turing-complete smart contract such as a subscription payment or time-based access. In this paper, we present both the multi-chain interface and a proof of concept implementation using the Algorand blockchain and their native digital asset tokens, Algorand Standard Assets (ASAs).

## 3 APPROACH OVERVIEW

Our approach can be broken down into two parts: the issuance of an authorization asset and the authentication of a resource user and their authorization asset. The asset defines the permissions that a user has when accessing a resource such as the type of subscription they have to a service, whether they can read certain data, write certain data, etc. Issuing the asset occurs once, while the verification of the asset occurs frequently (when obtaining a new session with the resource).

There are multiple methods of issuing the asset but the verification of an asset and user is almost identical for each of our proposed methods of issuance. Common across the issuance schemes is that they all produce an asset on a blockchain that the resource provider will use to determine what access the user has to its resources during the verification step. This is typically done by looking at the asset metadata. The metadata can be formed in any way that the asset creator would like, but note since it is stored on-chain, it is publicly viewable to anyone. Using hash algorithms or encryption algorithms is recommended for the metadata.

### 3.1 Issuance of Authorization Asset

We present three methods to distribute the authorization asset that will each guarantee the authorization asset being used to obtain access to resources is legitimate.

*3.1.1 Resource Creates.* In the case that the user does not mind owning an asset created by the resource provider, which is indicated in the asset, the resource provider can create the authorization asset. This case is ideal for resource providers with many user accounts with the same access permissions, and in this case access to the resource should be public (i.e. the user wants to show that they have an account with a specific resource provider).

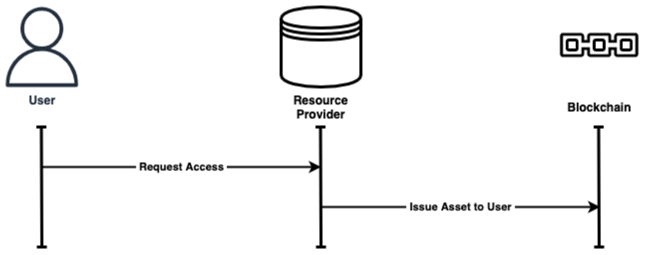


Fig. 1. The Authorization Process when the Resource Provider Creates the Asset

*3.1.2 User Creates.* In the case that the user wishes to remain anonymous, they can create the authorization asset using access data provided by the resource provider. This method creates a unique asset for each user of the service and imposes the cost of creating assets on the user. In this scenario the resource provider provides encrypted access data for the user to put into their asset to avoid fraud (if the data is not encrypted any user could create an asset with access permissions to any resource they want) and improve privacy (very difficult if not impossible to identify the resource provider or the resources the asset gives access to).

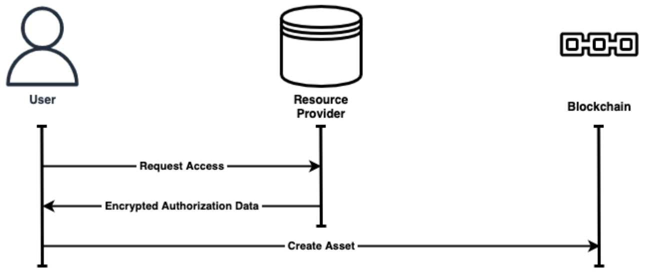


Fig. 2. The Authorization Process when the User Creates the Asset

*3.1.3 Smart Contract Creates.* This case is similar to the case in which the resource provider creates the asset but the user does not have to trust the resource provider to issue their asset. This method provides the same level of privacy as when a resource creates the asset but will not require the user to trust the service provider as much and can more closely couple a payment to the resource provider with the receipt of the asset.

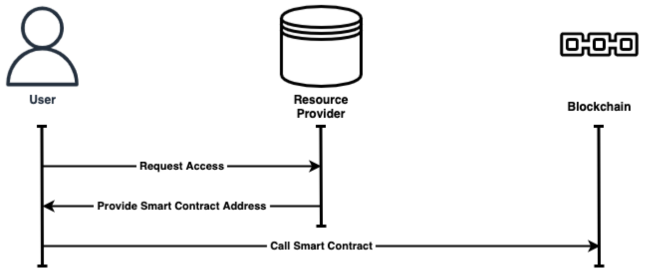


Fig. 3. The Authorization Process when a Smart-Contract Creates the Asset

### 3.2 Authentication Scheme

One important security consideration when granting access is to verify the user actually knows the private key for the public key that they claim to be. For our interface, we propose a challenge/response step where users will sign the challenge provided by the resource server with their private key. The signature is then checked and verified that it is valid by the resource server before proceeding with any other validity checks. In our interface, a mandatory validity check after validating the signature is the asset lookup on the blockchain.

For our interface, we will use the EIP-4361 standard (Sign-In With Ethereum Standard) which outlines important fields for the challenge such as domain, expirationDate, address, and more [[Rocco et al. 2021]] as seen below. The challenge groups all these fields into a formatted string message to be signed and sent back by the user. The user will see the challenge in plaintext, so they know exactly what they are signing and what permissions they are signing in with.

```
interface EIP4361Challenge {
    domain: string;
    address: string;
    statement: string;
    uri: string;
    version: string;
    chainId: string;
    nonce: number;
    issuedAt: string;
    expirationDate?: string;
    notBefore?: string;
    requestId?: string;
    resources?: string[];
}
```

*3.2.1 Our Changes to EIP-4361.* There are a couple of adjustments that were made to the EIP-4361 interface in order to make it suited for our specific implementation. First, Ethereum will not always be used, so we need to make it multi-chain compatible. The address field will need to be a valid address of whatever chain you are using, not always an Ethereum address. Similarly, the native signature and verification algorithms of the chosen blockchain will be used, not always Ethereum's. Second, we have decided to use the resources string array field for the asset ids encoded as strings instead of solely for an array of URI strings. This decision was made for the sake of not adding anything new to an existing, time-proven standard. We do potentially see a situation where URIs could need to be requested as resources as well as asset IDs, so we will make this flexible so that it supports both. We will prefix all asset IDs with 'Asset ID: '. Anything else not beginning with this prefix should be a validly formatted URI as defined in EIP-4361.

*3.2.2 Nonce Generation.* We have decided to use the current block (round) number as our nonce to be used within the challenge for our specific proof-of-concept implementation. Note that the nonce generation algorithm can be changed by any authorizing resource if they would like.

Although a number purely used once per account would be the most ideal, it introduces a lot of new overhead that we feel is not necessary for our implementation. A centralized backend database would have to keep stores of what numbers have been used and have not been. We envision our solution can be used in a completely decentralized way with no central server, so this data store would not be ideal.

Our proposed solution is to use the most recent block index as the nonce, and upon verification, we verify that a block with the requested index has been added to the main chain within the past minute. We do note that because technically the same block number can be used twice to verify if they were both verified within that minute span, it does not fully protect against man-in-the-middle attacks. For example, Bob signs the challenge and sends it via an insecure channel to the verifier; someone can intercept the challenge and also send it in for verification under Bob's address and get granted the same privileges.

It is important to protect against man-in-the-middle attacks as explained above. Every implementation of our interface should have some sort of defense against this, whether it is in the nonce generating step or the permission granting step. We argue that in our implementation, we adequately protect against this due to the minute time limit and our additional checks before granting permissions, which will only grant the first requester the permissions. Again, other implementations can define different nonce generation algorithms if preferred.

*3.2.3 Asset Lookup.* Once the user's identity as the owner of their public key has been verified through the digital signature of the challenge, Blockin can inspect the blockchain to verify the user's requested authorization asset is actually owned by them. Also, note that it is safe to assume that the wallet obtained this asset within the rules of the contract outlined upon asset creation due to the security offered by the blockchain and smart contracts.

Because authorization assets may be allowed to be transferable, this solution allows us to decouple identity and access. For example, if a rental car uses our proposed solution to provide fine-grained access to the vehicle, the rental company could loan one of their "unlock door" and "start engine" assets to a customer that has rented the vehicle. Once the customer is done with their rental, the rental company could retrieve their assets and the customer will no longer be able to access the rental car. The rental car itself doesn't consider whose public key is presenting the asset, only that it has the specific asset that allows access to those resources.

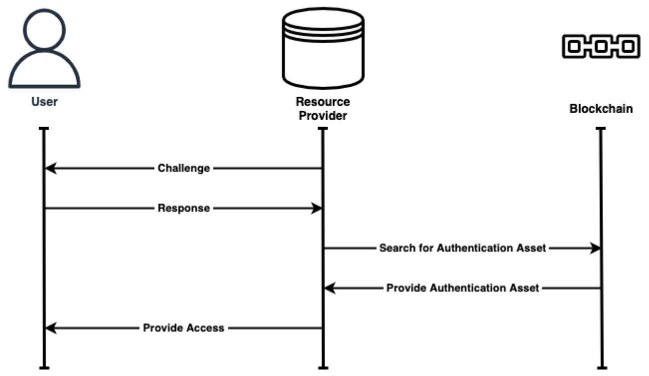


Fig. 4. The Authentication Process

## 3.3 Asset Encoding Scheme

While the provider can implement their own scheme, we suggest using the SecPAL authorization language [[Becker et al. 2010]]. When this is too large to store on chain, we recommend hashing the authorization data and storing a mapping of the generated hash to the authorization data in a database.

## 3.4 Hashing Authorizations for the User Creates Method

In order to prevent fraud when the user creates assets, we recommend to prefix the authorization permissions with the user's address and a secret generated by the site before hashing it with the SHA256 algorithm. When the user creates an asset with this hash, the asset's unique identifier should be stored in a database with the raw permissions data. The inclusion of the user's address guarantees that they did not copy an existing user's access permissions hash. The inclusion of the secret guarantees that the user did not manipulate the permissions encoding scheme to provide themselves with access to whatever they want. The user can still transfer this authorization asset to any account on the chain because the original creator's address will always be publicly visible.

## 4 IMPLEMENTATION REQUIREMENTS

In this section, we will discuss the core technical requirements for implementing our design, and then we will discuss more specific technical aspects that are not required but highly desirable to build the best solution. In its simplest form, our solution relies on a decentralized application (**dApp**) and a blockchain. We will first discuss the dApp and then the chain.

### 4.1 dApp

*4.1.1 Required Features.* The required technical aspects for the dApp are very simple. As discussed in previous sections, the resource provider presents the resource owner with a challenge to be signed using the private key of the wallet holding the asset. The dApp serves as the mechanism for the resource owner to prove their identity, and therefore the dApp must be able to establish a connection with the wallet in order to sign the challenge. Given a dApp that can establish a connection with a wallet, this is enough to prove the resource owner's identity.

*4.1.2 Desired Features.* While it is enough to have a dApp that can merely connect with a wallet to handle the challenge / response process, there are several additional aspects that can significantly improve the quality of the dApp as it interacts in this design.

Most obvious, the connection between the dApp and the wallet should be as secure as possible. If the private key or any communication data between the dApp and the wallet is somehow exposed, this could introduce attack surfaces that would compromise the security of this design.

It would also be ideal for the dApp to support as many devices as possible. Authorization is an aspect of practically all modern devices, and therefore the design we propose could benefit a wide range of technologies. However, certain approaches could greatly limit the compatibility of our dApp with certain devices, such as if the dApp was written in Swift and could only run on iOS. Similarly, if the dApp assumed the device has access to a significant amount of resources, this could hinder support for light-weight clients like smart watches and other IoT devices. A great solution would be one that minimizes resource requirements and maximizes cross-platform support so that the dApp can support as many technologies as possible.

The dApp should also be as accessible as possible. Without access to the dApp, there is no way for the resource owner to achieve authorization. Therefore, denial-of-service attacks are a notable threat to the dApp that must be addressed. Scalability is also a very important issue. As mentioned before, a wide array of devices could benefit from this solution, and therefore it should be anticipated that the dApp will be responsible for facilitating in the authorization of potentially billions of devices simultaneously. Under this same assumption, the dApp should also be as cost-efficient as possible.

### 4.2 Blockchain

*4.2.1 Required Features.* At its core, the chain must support the creation of assets with enough metadata to describe the permissions associated with the access token. Furthermore, wallets on the chain must be able to establish a connection with the dApp and support a native message signature algorithm. Finally, recall from the previous section the third approach in which a smart contract creates the asset. Clearly, this approach assumes that it is possible to deploy custom smart contracts on the chain. Therefore the third approach also requires support for smart contracts, but this is not necessary for the other two approaches.

*4.2.2 Desired Features.* First, an ideal blockchain is one that adequately solves the Blockchain Trilemma of security, scalability, and decentralization. In regards to security, if the chain is compromised this will in turn compromise the security of our design. Therefore, assets on the chain should be protected so that access tokens cannot be stolen, deleted, or otherwise tampered with by unauthorized parties. By extension, this also means that the private key of wallets on the chain should never be compromised or predicted by unauthorized parties.

As mentioned before, it should be anticipated that this approach will service potentially billions of devices simultaneously. Consequently, asset creation, lookup, and transfer should be highly scalable. Ideally, an asset can be transferred an unlimited number of times and there should be no restriction on the number of assets and smart contracts a resource creator can create and deploy, respectively.

Although this approach could work in a centralized ecosystem, we believe the greatest value of our approach stems directly from its decentralization. Through decentralization, our approach is able to remove trust from the authorization process. In Web 2.0, authorization requires trust in some centralized database to confirm the identity and privileges of an individual. By relying on a decentralized blockchain in place of a centralized database, this shifts the attack surface and presents an alternative approach for authorization which we believe to be more secure.

It must also be acknowledged that certain blockchains are more expensive than others. Greater security is worth a certain level of cost, but ideally that cost should be minimized. Therefore, the less expensive it is to create and transfer access tokens the better. Similarly, if smart contracts are used in the design, the cost to deploy and execute a smart contract should also be minimized.

Moreover, it must also be acknowledged that certain blockchains process transactions and achieve finality at a faster rate than others. If an access token can be created or transferred in minimal time, this will minimize the total time required to create access tokens and transfer them from one wallet to another. The same applies to the deployment and execution of smart contracts. In all cases, this contributes directly to the speed and corresponding convenience of this authorization design.

Another important aspect is the developer ecosystem of the blockchain. Blockchains that are open-source with strong documentation, community, and developer tools will prove superior. This will ensure the quality, support, and longevity of our proposed authorization approach.

Finally, if the blockchain provided the ability for resource creators to freeze or revoke a resource owner's access token, this could further increase the potential applications of our authorization approach.

## 5 ALGORAND

Using the required and desired features described in the previous section as criteria for selecting a blockchain to develop an example implementation of our idea, we decided to build on Algorand. We believe that Algorand's unique features are a perfect fit for our interface and digital authorization assets. In this section, we will elaborate on why. However, note that this Algorand was just used as a proof-of-concept, our interface can support any blockchain.

### 5.1 Algorand Standard Assets

Algorand Standard Assets (ASAs) refers to the assets on Algorand's network. Similar to other standard assets like Ethereum's ERC-721 token, ASAs can be used to create NFTs. However, ASAs provide some additional valuable features such as the ability for the ASA creator to specify a 'freeze' and 'clawback' address, where the freeze address can freeze and unfreeze an ASA holder's ability to transfer their asset and the clawback address can transfer the ASA from the current holder to another address. ASA creators can also specify a whitelist of permitted addresses that the asset can be sent to and from. Conveniently, the freeze address, clawback address, and whitelist addresses are all mutable values [[Algorand 2021c]]. As it will be shown later in this paper, these are very attractive features that lend themselves nicely to our approach.

### 5.2 Cost

Unlike Ethereum and other blockchains that are very expensive due to poor network scalability, creating assets on Algorand is incredibly cheap. An account's minimum balance to create an ASA begins at 0.1 Algos (their native currency) and increases by 0.1 Algos with each new ASA created [[Algorand 2021c]]. At the time of this writing, 0.1 Algos is approximately 8 cents in US dollars.

### 5.3 Speed

Traditional Proof-of-Work (PoW) blockchains such as Bitcoin and Ethereum have proven to suffer from very slow transaction speeds, but many PoS blockchains also suffer from reduced security. Algorand uses a Pure Proof-Of-Stake (PPoS) consensus mechanism, benefiting from the speed of PoS networks while also using Verifiable Random Functions (VRFs) and Cryptographic Sortition to randomly and secretly select users to participate in the consensus protocol [[Algorand 2021d]]. Additionally, Algorand's blockchain never forks, and as a result it achieves immediate transaction finality. When a new block is added to the chain it is guaranteed to be permanent, and this significantly reduces the total time needed to confirm that a transaction was successfully recorded on the blockchain [[Algorand 2021e]].

### 5.4 Scalability

As mentioned in the previous section, it is important that the blockchain used is highly scalable for this implementation to have a wide and diverse range of potential applications. Aside from the fast speed and low cost, Algorand also helps users to create assets and deploy smart contracts on a large scale. Users are allowed to create unlimited ASAs as of March 2, 2022, and the Algorand team is actively working to allow for opting into and deploying unlimited smart contracts [[Algorand 2022]]. Additionally, through the use of Atomic Transfers in Layer-1, ASA creators can transfer multiple assets to multiple addresses as a group such that either all transactions are processed or none are processed [[Algorand 2021a]]. This feature improves the security and efficiency of managing the ASA transactions at scale.

### 5.5 Rekeying

In many traditional blockchains, it is impossible to change the private key of a wallet without also changing its public address. In the future, if people rely on these assets as utilities in their daily lives, it could be very computationally burdensome and wasteful to have to transfer all assets to a new address in the event that the user accidentally exposes their private key. Algorand solves this problem by providing the ability for users to change their private key without needing to create a new public address in Layer-1 [[Algorand 2021f]]. Looking forward to the future, this will become an increasingly important issue as a broader audience of society begins to adopt these technologies.

### 5.6 Support

It is important to build with a blockchain network that has strong documentation and strong language support. Unlike many other blockchains, Algorand has strong documentation and supports Turing-Complete languages (specifically Python), as well as Reach [[Algorand 2021b]]. This will provide for a more efficient and secure development environment.

## 6 IMPLEMENTATION

Our design and approach were implemented via a JavaScript library that we plan to publish as an NPM (Node Package Manager) library. This JavaScript library provides functions needed for both the user and the resource server at all stages of the authorization process.

In addition to the library, we have created a demo dApp (decentralized application) site that uses our library to authenticate users on the frontend. Within this site, we also show how to use our library to authenticate users with a backend resource server as well.

### 6.1 Javascript Library

As mentioned above, the JavaScript library is the core library that handles everything. It can currently be found at https://github.com/matt-davison/blockin/.

Our library handles the issuance of the digital authorization assets as well as the challenge / response generation and verification. The scope of our library was kept narrow on purpose. It focuses solely on authorization and thus can be easily integrated with any frontend or backend codebase.

Note that one thing that is out of scope for this library is signing the transactions with a valid wallet provider. Our library aims to be flexible to integrate with any codebase and across many chains. We did not want to be limited to a single wallet provider, so we decided to leave the digital signatures with secret keys out of our library's functionality. For everything needing a signature while interacting with the Blockin library, users will receive something to sign from the Blockin library, go to any arbitrary wallet provider, sign it, and return that signature to the Blockin library. This is to protect user security. Blockin does not need and will never ask for your private key. Blockin verifies everything using digital signatures which are outsourced to wallet providers or whoever holds the users' private key.

Lastly, we have currently only implemented our library to support Algorand and Algorand Standard Assets, but as shown previously in this paper, the Blockin interface is flexible. We have designed the library in such a way that we have decoupled chain-specific functions. For anyone who wishes to apply our interface to another chain, such as Ethereum, all they will have to do is rewrite the chain-specific functions to be handled using Ethereum's native methods.

The functions, as explained in our Approach Overview section, exported from our library for clients to use can be categorized into two parts: Issuance of Authorization Assets and Challenge / Responses. We will now dive deeper into the inner workings of these functions.

*6.1.1 Issuance of Authorization Assets.* The asset authorization part of the library provides all the functionality needed to implement the three methods of creating authorization assets shown previously: user creates, resource creates, and smart contract creates. Since all blockchains will have different ways of creating and transferring assets, we attempted to make this as flexible for any chain as possible, which we will explain in a later section. We attempted to abstract everything into functionality needed by all chains. We came up with the following functions that will be implemented by any chain. The opt in transaction may be left blank by chains that don't require this (required in Algorand).

```
createAssetTxn
createAssetOptInTxn
createAssetNoOpTxn
createAssetTransferTxn
sendTxn
```

Either the user or resource will call the above functions depending on the asset creation method choice. Note that Blockin provides this part of the library for convenience. Assets don't have to be created using these functions specifically. One can create them via another library, or if they are already created, these functions don't need to be used.

*6.1.2 Asset Creating Smart Contract.* Above, we showed how to do the user creates and resource creates methods. For the smart contract creates method, we provide a sample and template using a smart contract on Algorand created using PyTeal. This can be found in the smart-contracts folder. Again, this is provided for convenience. Smart contracts can be created via any method a developer wishes. Due to Algorand requiring users to opt-in to assets before they may be transferred to them, this smart contract requires a few steps in order for an authorization asset to be created and distributed to a user.

(1) A user must opt-in to the smart contract. Our implementation utilizes local state, and thus requires the user to opt-in to the smart contract as well, but this can be replaced by using the global state of the smart contract and storing mappings of user address to asset ID.
(2) Second, the resource provider calls the smart contract and provides the authorization data according to their schema and the user's address. The smart contract then creates the asset and places the created asset's ID in the user's local state.
(3) Third, the user must opt-in to the created asset's ID so that they may retrieve it in the next step.
(4) Finally, The user calls the smart contract, and the smart contract uses the asset ID stored in their local state to send the matching asset from the smart contract's holdings to the user, completing the issuance of the user's authorization asset.

*6.1.3 Challenge Creation.* Users will call createChallenge() to construct the message string to be signed as explained in the Authentication Scheme Section. As a reminder, we will be using the EIP-4361 Sign In With Ethereum specifications with two minor modifications. First, the nonce number generated will be a recent block hash / id number. Note that this is Algorand implementation specific and can be changed if needed. Second, the resources field will be asset IDs encoded as strings instead of solely URIs. Below, we show the library TypeScript code that creates a challenge. Users will call createChallenge() and be returned a valid EIP-4361 string to sign if the function doesn't throw an error. They will then sign this string using their wallet provider and send both the signature and the message to the authorizing party, as explained in the next section.

```
export async function createChallenge(
    domain: string,
    statement: string,
    address: string,
    uri: string,
    expirationDate?: string,
    notBefore?: string,
    resources?: string[]
) {
    try {
        const challenge: EIP4361Challenge = {
            domain,
            statement,
            address,
            uri,
            version: "1",
            chainId: "1",
            nonce: await getChallengeNonce(),
            issuedAt: new Date().toISOString(),
            expirationDate,
            notBefore,
            resources
        }

        validateChallenge(challenge);

        return constructMessageString(challenge);
    } catch (error: unknown) {
        return `Error: ${error}`;
    }
}
```

An example created challenge is provided below. Note that some line breaks are applied here for formatting purposes.

```
    https://blockin.com wants you to sign in
    with your Algorand account:
    NLQKOUWSN6I5I4N4W7HQAWVUIRDS5JQPHNEPQ7CO-
        KR5YTAFQMHDEKIJJ4Q

    Sign in to this website via Blockin.
    You will remain signed in until you terminate
    your browser session.

    URI: https://blockin.com/login
    Version: 1
    Chain ID: 1
    Nonce: 21262114
    Issued At: 2022-04-28T20:41:26.652Z
    Expiration Time: 2022-05-22T18:19:55.901Z
    Resources:
    - Asset ID: 85934209
```

*6.1.4 Challenge Verification.* The authorizing party is responsible for calling verifyChallenge(). Users will submit both the challenge generated from createChallenge() and the signed challenge to the authorizing party. The function below will then be called by the authorizing party. If the function returns without an error, they can safely authenticate the user using whatever method they prefer (JWTs, session tokens, etc.). This function has three parts to it: verifying inputted challenge is well-formed, verifying the signature is correct, and verifying the address actually owns the requested assets in their wallet. The signature verification and asset verification are both implementation-specific for whatever chain you are using, so they are implemented for Algorand in our library's case. Lastly, we provide a grantPermissions() function where an implementation can implement permissions such as granting cookies, JWTs, session tokens, etc. We left this blank for our implementation, but it is there to show how it can be used, if needed.

```
export async function verifyChallenge(
    originalChallenge: Uint8Array,
    signedChallenge: Uint8Array
) {
    try {
        const generatedEIP4361ChallengeStr: string =
          await getChallengeString(originalChallenge);

        const challenge: EIP4361Challenge =
            createMessageFromString(
                generatedEIP4361ChallengeStr
            );

        validateChallenge(challenge);
        console.log("Success: Constructed challenge
          from string and verified it is well-formed.");

        const originalAddress = challenge.address;
        await verifyChallengeSignature(
            originalChallenge,
            signedChallenge,
            originalAddress
        )
       console.log("Success: Signature matches address
            specified within the challenge.");

        if (challenge.resources) {
            await verifyOwnershipOfAssets(
                challenge.address,
                challenge.resources
            );
          await grantPermissions(challenge.resources);
        }

      return `Successfully granted access via Blockin`;
    } catch (error) {
        return `Error: ${error}`;
    }
}
```

*6.1.5 Library Abstraction.* Because our library aims to be multi-chain and flexible, we designed our library in a way that supports this. We did this by designing a ChainDriver interface that defines the functions needed by every blockchain such as sendTxn(), makeAssetTxn(), or isValidAddress(). Anytime the library needs to do something chain-specific such as calling the chain's block indexer or getting the currently recommended transaction parameters, this is done via the implementation of the ChainDriver interface passed into Blockin's initialization function, setChainDriver().

For the purpose of our demo dApp, we created an AlgoDriver implementation that is specific to the Algorand blockchain, which can be viewed at https://github.com/matt-davison/blockin/blob/main/src/ChainDrivers/AlgoDriver.ts. However, this interface allows for the flexibility of any chain to be implemented and supported on Blockin. For example, if a new blockchain comes along, all a developer will have to do to integrate it within Blockin is to create a class that implements this interface for their blockchain of choice. We plan to offer multiple chain drivers already implemented, such as AlgoDriver, directly from the Blockin library as imports for convenience.

```
export interface IChainDriver {
    getChallengeStringFromBytesToSign:
        IGetChallengeStringFromBytesToSign,
    makeAssetTxn: IMakeAssetTxn,
    makeAssetOptInTxn: IMakeAssetOptInTxn,
    makeAssetTransferTxn: IMakeAssetTransferTxn,
    sendTxn: ISendTx,
    getLastBlockIndex: IGetLastBlockIndex,
    getAllAssetsForAddress: IGetAssets,
    getTimestampForBlock: IGetTimestampForBlock,
    isValidAddress: IIsValidAddress,
    getPublicKeyFromAddress: IGetPublicKey,
    getAssetDetails: IGetAssetDetails,
    lookupTransactionById: ILookupTransactionById,
    verifySignature: IVerifySignature,
    verifyOwnershipOfAssets: IVerifyOwnershipOfAssets,
}
```

From the backend or frontend where you are calling Blockin from, it is very simple to define the chain driver. Here is how we specified which chain driver to use in our demo site implementation.

```
import { AlgoDriver, setChainDriver } from "blockin";

setChainDriver(
    new AlgoDriver(process.env.ALGO_API_KEY)
)
```

If a developer has created or imported their own chain driver that they would like to use with the Blockin library, they can simply tell Blockin to use this new chain driver as follows.

```
import { setChainDriver } from "blockin";
import { CustomChainDriver } from "./customChainDriver"
// CustomChainDriver extends the IChainDriver interface

setChainDriver(
    new CustomChainDriver(process.env.CHAIN_API_KEY)
);
```

In this way, our library provides strong flexibility for developers to create and use chain driver implementations for other blockchain networks not currently supported natively in the Blockin library.

## 6.2 Demo Site

We are also developing a demo site that can currently be found at https://blockin.vercel.app/, with the source code located at https://github.com/kking935/Blockin-Demo that uses our library to authenticate and authorize users. This website is a dApp (decentralized application) which uses our library as a dApp tool to provide these authentication and authorization features for users on the frontend. We also show on this demo site how to authenticate and authorize users for a backend resource server as well.

This demo site shows everything about our library from creating the authorization asset in all three ways (resource creates, user creates, and smart contract creates) to create / verify challenges and responses.

*6.2.1 Cookies.* For our demo site, we decided to implement our permissions in the form of a cookie called 'blockedin'. When verifyChalllenge() is called and succeeds, the site will create a session cookie called 'blockedin' with a value of the asset ID requested and store it in the browser's local storage. The cookie value can then be used to grant users access with role-based access. As shown in the Sign In Privileges section below, different cookie values can correspond to different access privileges. Browser cookies were just our choice for granting permissions. Other implementations may choose to use JWTs, HTTP-only cookies, or any other method.

*6.2.2 Sign In Privileges.* To demonstrate how custom asset metadata hashes correspond to different role-based access controls, we change the color of the navigation banner at the top of the page based on the asset metadata. The site supports any valid HTML color. The metadata hash stored on chain is calculated as Base64(SHA256(HTML color plaintext)). The site then inverts this operation when attempting to determine the color of a requested asset after verifyChallenge() succeeds.

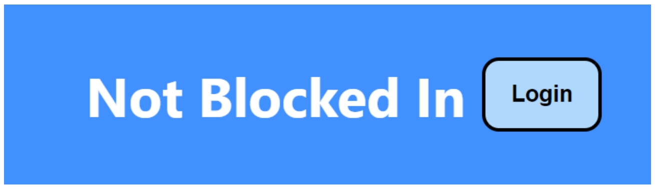


Fig. 5. Banner - Not Logged In

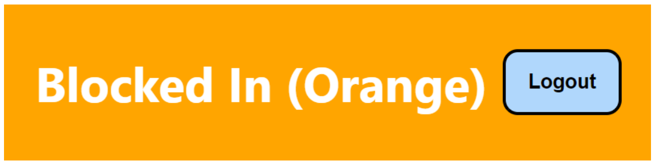


Fig. 6. Banner - Logged In with 'Orange' Asset Metadata Encoding

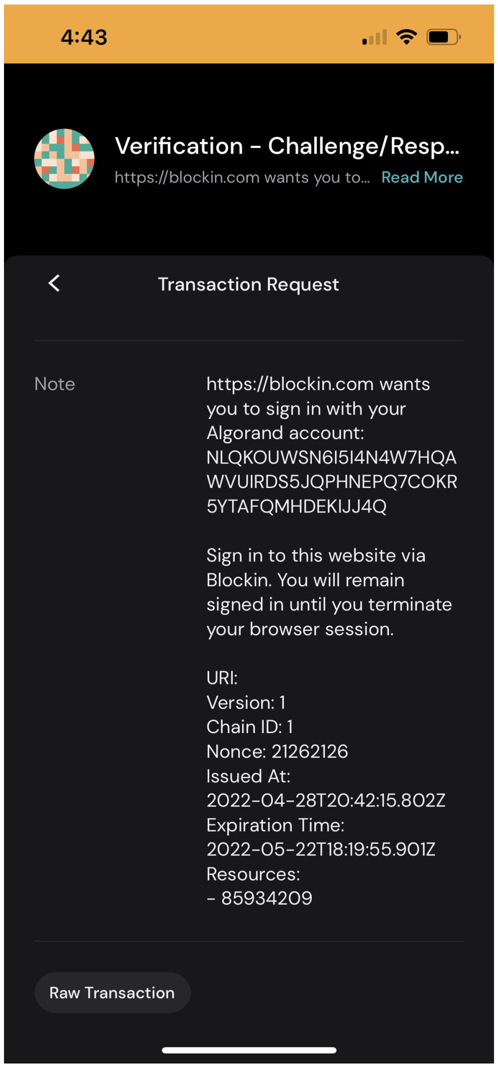
*6.2.3 WalletConnect.* A challenge we faced when developing our demo site was interacting with a user's blockchain accounts. We chose to use WalletConnect in order to facilitate initiating a connection between the user's browser and their wallet of choice [[WalletConnect 2022]].

*6.2.4 Backend Demo.* We use a hybrid approach in our demo site to honor a best practice of not leaking any API keys directly in the browser. Almost all the functions called by Blockin have to trigger some API for a specific chain such as to get transaction parameters, asset details, or to send a transaction. If there is no worry about API keys leaked or if one implements the ChainDriver interface such that it doesn't need to call any API, then the Blockin library can be called directly from the frontend code; however, this may introduce other security vulnerabilities.

As mentioned, we have a private Algorand API key, so the demo site makes use of the following flow of events: the frontend (user) will call the created backend (authorizing resource) for all sign-in related requests. If needed, the backend will call the Blockin library, using its private API key. We envision this will be the typical flow for most implementations using Blockin. This also allows for the authorizing resource more flexibility to provide specialized tokens such as custom JWTs for different privileges or HTTP-only cookies for access to their backend.

This API can be found in the pages/api folder of our demo code. We abstracted it so that every time we call Blockin, it is done via the backend server. The following routes were provided in the demo, but note this is just for our custom implementation.

```
/api/createAssetTxn
/api/createOptInTxn
/api/getAssetDetails
```


4:43
Verification - Challenge/Resp...
https://blockin.com wants you to... Read More
Transaction Request
Raw Transaction

Note

https://blockin.com wants you to sign in with your Algorand account:
NLQKOUWSN6I5I4N4W7HQAWVUIRDS5JQPHNEPQ7COKR5YTAFQMHDEKIJJ4Q

Sign in to this website via Blockin. You will remain signed in until you terminate your browser session.

URI:
Version: 1
Chain ID: 1
Nonce: 21262126
Issued At:
2022-04-28T20:42:15.802Z
Expiration Time:
2022-05-22T18:19:55.901Z
Resources:
- 85934209

Fig. 7. Wallet Connect Challenge Signature Mobile Example - Pera Wallet

```
/api/getAssets
/api/getChallenge
/api/getToken
/api/receiveToken
/api/sendTxnToNetwork
/api/verifyChallenge
```

## 7 EVALUATION

We have decided to compare our Algorand proof-of-concept implementation of our interface to two other implementations that support micro-authorizations: existing Web 2.0 OAuth solutions and existing Ethereum ERC-721 access token solutions. Our criteria for comparing the solutions are cost, speed, scalability, decentralization, and ease of use. When we use "Blockin", we are referring to our proof-of-concept on Algorand.

Also, we would like to note that our solution inherits from and is very similar to EIP 4361 - Sign in With Ethereum. However, Blockin has all the functionality that Sign in with Ethereum has and more. This is due to it supporting multiple chains natively as well as the flexibility of requesting specific assets.

### 7.1 Cost

These are estimates based on network statuses and exchange rates at the time of writing.

- Blockin
  - $0.001 or 0.001 Algos fees to create asset (one-time)
  - $0.001 or 0.001 Algos fees to opt in (one-time)
  - $0.001 or 0.001 Algos fees to transfer asset
  - 0 fees to sign challenge / response each time
- Ethereum
  - Up to 0.3 ETH or $1000 fees to create asset (one-time)
  - No opt in transaction
  - Up to 0.2 ETH or $500 fees to transfer an asset
  - 0 fees to sign challenge / response each time
- OAuth 2.0
  - $100 - $200 fees per month to run a server for an average business

### 7.2 Speed

- Blockin
  - 4.4 seconds to create asset (one-time)
  - 4.4 seconds to transfer asset
  - HTTP request / response speeds for challenge / response (every time)
- Ethereum
  - 72 seconds on average to create asset (one-time)
  - 72 seconds to transfer asset
  - HTTP request / response speeds for challenge / response (every time)
- OAuth 2.0
  - HTTP request / response speeds plus database lookup speeds (every time)

### 7.3 Scalability

- Blockin
  - Up to 1300 transactions per second
  - Assets are a native, lightweight feature built into Algorand which doesn't require any expensive smart contracts to be created
- Ethereum
  - Up to 15 transactions per second
  - Requires an expensive smart contract for each asset
- OAuth 2.0
  - Practically infinitely scalable with the size of the database

### 7.4 Decentralization

- Blockin and Ethereum
  - Decentralized by many independent nodes working together to validate the state of the chain
  - No tampering of data

- OAuth 2.0
  - Not decentralized at all
  - Must run a central server
  - Not natively compatible with Web 3.0 public / private key pairs
  - Metadata can potentially be tampered with by a centralized party

### 7.5 Ease of Use

Ease of use is also important to consider in our comparison. OAuth 2.0 has been the industry standard for a while, and many are used to it. However, you have to remember your usernames and passwords for each website.

Ethereum and Algorand both offer wallet providers that allow you to have a single sign-on password (your private key) for any website. We believe that Algorand is more usable due to its transaction finality speed and mobile compatibility.

### 7.6 Lightweight

We also want to acknowledge that the total size of our library is currently only 34 KB which is lightweight and makes it easy to integrate.

## 8 CONCLUSION

In this paper, we introduced Blockin. Blockin is a multi-chain, flexible sign-in standard that supports dynamic role-based access control, expiration tokens, and micro-authorizations through the use of digital assets and their metadata stored on any blockchain.

The reason we decided a framework like Blockin was needed was two-fold. First, we observed that OAuth 2.0 had lots of functionality in Web 2.0 that hasn't yet transitioned to Web 3.0 with the new paradigm of every user owning public-private key pair. Second, we also noted that a multi-chain standard has not been proposed for Web 3.0 sign-ins. There have been specific chain implementations, such as Sign-In with Ethereum. However, not every user who wants to sign in to Netflix, for example, will have an Ethereum address. Users will all have their own preferred chains.

Our goal was to combine the previously mentioned solutions, other Web 3.0 technologies (such as smart contracts and digital assets), and the existing public/private key standard to provide a multi-chain, general interface for the functionality needed for role-based access, expiring tokens, and dynamic resource privileges. In particular, we showed how to improve upon the speed, scalability, cost, and flexibility of the existing solutions while also introducing new features such as role-based access, freezing, expirations, and clawbacks of tokens.

To implement Blockin, we built a JavaScript library that can be categorized into two parts. First, the asset creation can be performed using three different methods as defined in this paper: user creates, resource creates, and smart contract creates. Each method has its pros and cons and different execution flows, but all result in an asset with metadata being created on-chain that can be used with Blockin. Second, the verification part of the library focuses on the challenges and responses needed for authorizing resources to verify a user's sign-in request. For this, we decided to inherit the EIP-4361 Sign In with Ethereum standard, with a couple of minor modifications. Users will submit a challenge with the requested assets they want to sign-in with. Blockin will then verify the challenge was well-formed, signed correctly, and verify the requested user actually owns the requested assets on-chain. If this verification challenge check succeeds, micro-authorizations can be granted based on the specific asset and its metadata.

We then showed why Algorand was our preferred choice for implementing our library's proof-of-concept due to its unique asset features (freezing and clawbacks) and blockchain features (low cost, high speed, high scalability). Lastly, we explored the inner workings of the library and showed a demo example of how a site would use Blockin.

We believe Blockin has huge potential to become a user-centric, universal sign-on standard for Web 3.0. We plan to continue to maintain and expand the open-source JavaScript library functionality in the future.

## REFERENCES

Mustafa Al-Bassam. 2017. Scpki. *Proceedings of the ACM Workshop on Blockchain, Cryptocurrencies and Contracts* (2017). https://doi.org/10.1145/3055518.3055530

Algorand. 2021a. Algorand Atomic Transfers. https://developer.algorand.org/docs/get-details/atomic_transfers/

Algorand. 2021b. Algorand Smart Contracts. https://developer.algorand.org/docs/get-details/dapps/smart-contracts/

Algorand. 2021c. Algorand Standard Assets (ASAs). https://developer.algorand.org/docs/get-details/asa/

Algorand. 2021d. Frequently Asked Questions. https://www.algorand.com/technology/faq

Algorand. 2021e. Immediate transaction finality. https://www.algorand.com/technology/immediate-transaction-finality

Algorand. 2021f. Rekeying. https://developer.algorand.org/docs/get-details/accounts/rekey/

Algorand. 2022. Algod : Merge Unlimited Assets to master. https://github.com/algorand/go-algorand/pull/3652

Seyed Mojtaba Bamakan, Nasim Nezhadsistani, Omid Bodaghi, and Qiang Qu. 2022. Patents and intellectual property assets as non-fungible tokens; key technologies and challenges. *Scientific Reports* 12, 1 (2022). https://doi.org/10.1038/s41598-022-05920-6

Moritz Y. Becker, Cédric Fournet, and Andrew D. Gordon. 2010. SecPAL: Design and semantics of a decentralized authorization language. *Journal of Computer Security* 18, 4 (2010), 619–665. https://doi.org/10.3233/jcs-2009-0364

Nikos Fotiou, Iakovos Pittaras, Vasilios A. Siris, Spyros Voulgaris, and George C. Polyzos. 2020. OAuth 2.0 authorization using blockchain-based tokens. *Proceedings 2020 Workshop on Decentralized IoT Systems and Security* (2020). https://doi.org/10.14722/diss.2020.23002

Priyanka Kamboj, Shivang Khare, and Sujata Pal. 2021. User authentication using blockchain based smart contract in role-based access control. *Peer-to-Peer Networking and Applications* 14, 5 (2021), 2961–2976. https://doi.org/10.1007/s12083-021-01150-1

Gregory Rocco, Wayne Chang, Nick Johnson, and Brantly Millegan. 2021. EIP-4361: Sign-in with Ethereum. https://eips.ethereum.org/EIPS/eip-4361

WalletConnect. 2022. WalletConnect. https://walletconnect.com/

Yuxin Zhong, Mi Zhou, Jiangnan Li, Jiahui Chen, Yan Liu, Yun Zhao, and Muchuang Hu. 2021. Distributed blockchain-based Authentication and Authorization Protocol for Smart Grid. *Wireless Communications and Mobile Computing* 2021 (2021), 1–15. https://doi.org/10.1155/2021/5560621